# Mining Inline Cache Data to Order Inferred Types in Dynamic Languages

Nevena Milojković[a,*], Clément Béra[b], Mohammad Ghafari[a], Oscar Nierstrasz[a]

[a]*Software Composition Group, University of Bern, Switzerland*
[b]*RMOD-INRIA Lille Nord Europe, France*

## Abstract

The lack of static type information in dynamically-typed languages often poses obstacles for developers. Type inference algorithms can help, but inferring precise type information requires complex algorithms that are often slow.

A simple approach that considers only the locally used interface of variables can identify potential classes for variables, but popular interfaces can generate a large number of false positives. We propose an approach called *inline-cache type inference* (ICTI) to augment the precision of fast and simple type inference algorithms. ICTI uses type information available in the inline caches during multiple software runs, to provide a ranked list of possible classes that most likely represent a variable's type. We evaluate ICTI through a proof-of-concept that we implement in Pharo Smalltalk. The analysis of the top-$n$+2 inferred types (where $n$ is the number of recorded run-time types for a variable) for 5486 variables from four different software systems shows that ICTI produces promising results for about 75% of the variables. For more than 90% of variables, the correct run-time type is present among first six inferred types. Our ordering shows a twofold improvement when compared with the unordered basic approach, *i.e.,* for a significant number of variables for which the basic approach offered ambiguous results, ICTI was able to promote the correct type to the top of the list.

*Keywords:* type inference, dynamically-typed languages, inline caches

DOI: 10.1016/j.scico.2017.11.003

*Corresponding author
*URL:* scg.unibe.ch/staff/Milojkovic (Nevena Milojković)

## 1. Introduction

Static type information has shown to be of crucial importance to developers during software maintenance [1, 2]. Inferring type information from source code in dynamically-typed languages has been extensively researched over the past decades [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. While some approaches rely only on the available static information [3, 4, 13, 8], others use dynamic execution to collect run-time type information and feed it back to the algorithm [14, 5, 10].

For static type analysis to be precise, it must closely track control and data flow. However, reliable results are usually achieved by analysing the whole program which is very expensive. Besides, modern software systems not only depend heavily on libraries, but are often part of a distributed system which may not be available for analysis. Simpler type inference analyses [8, 4] statically track variable assignments and the set of messages sent[1] to a variable in order to determine which classes either implement those methods, or inherit them from a superclass. Since they are neither control- nor data-flow sensitive, these approaches tend to be less precise, but very fast. Nevertheless, the problem with these simple approaches in dynamically-typed languages is that they provide a developer with the set of unordered classes that represent possible type for a variable. Unfortunately, this leaves the burden of looking for the correct type on the developer. To alleviate this issue, we have investigated a way of ordering the results of a simple type inference algorithm by statically analysing the frequency of class usages and class instantiations in the source code [15]. In that work, the focus was only on static data, which results in missing certain types when dynamic class loading or reflection are used [16].

Nowadays many virtual machines for dynamic languages include Just-in-Time compilers that use inline caches [17, 18] to achieve high performance. Inline caches have been exploited for compiler optimisation purposes [14]. Besides the information about methods that were previously selected to respond to a message send, these caches also contain receiver type information for the message send, which could be easily exploited in order to improve current development tools. This run-time information about the type of the receiver has already been used to feed the type back to code, and in case of successful type checking at run time, to inline the message send, and execute code faster. However, to the best of our knowledge, it has still not been used to improve static type information

[1] The terms "message" and "method" originate from Smalltalk, where one "sends a message" to an object, and the receiver then selects a "method" to respond to that message.

for other message sends, for which the receiver type has not been collected from inline caches. We believe that this information collected during execution of any program written in the same language would add productively to the statically collected knowledge used for inferring a variable's type. As run-time information has been read from the virtual machine, no instrumentation is required.

We present an approach called *inline cache type inference* (ICTI) to exploit type information collected from inline caches during program runs from different systems written in the same language in order to improve static type inference. We employ a simple static analysis algorithm to infer types of variables. Type information collected from inline caches during execution of other programs written in the same language is then used to order the types of these variables. This means that the possible classes for a variable are ordered based on the class usage frequency during program runs.

We have implemented a proof-of-concept prototype for Pharo[2], a dialect of Smalltalk, a highly reflective dynamically-typed language, which enables fast and easy implementation of analysis tools [19]. We have used this implementation to evaluate our claim that the frequency of class usage as the type of a receiver at run time can serve as a reliable proxy to statically identify the type of a variable. We have used a basic static analysis algorithm, *i.e.,* RoelTyper [8], to collect static type information based on the messages sent to variables and from assignments to them. We then augmented the results with the help of the inline cache information. The results show that the implemented heuristic is reasonably precise for more than 75% of the variables, and compared to the basic algorithm, ICTI more than doubled the number of correctly guessed types for a variable. We believe the improvement achieved by our heuristic can boost the performance of simple static type inference algorithms, regardless of their various applications in the field.

This article extends our previous work [20] as follows: (i) we present a motivating example for ICTI, (ii) we provide a thorough discussion of related work, (iii) we explain in detail how we gather type information from the runtime, (iv) we improve ICTI with a heuristic that collects type hints from method argument names, (v) we evaluate ICTI on 15% larger set of variables, (vi) we compare ICTI with the basic algorithm, and (vii) we discuss the results.

**Structure of the Paper.** We start by giving an overview of the problem in section 2. We discuss the related work in the field in section 3. Section 4 explains

[2] http://www.pharo.org Pharo is a Smalltalk IDE, including a large library that contains the core of the Smalltalk system itself.

the virtual machine used for run-time data collection. Next we define the used terminology and the implemented heuristic in section 5. Section 6 shows results of the evaluation of the prototype. We then describe potential threats to validity in section 7 before concluding in section 8.

## 2. Motivation

To explain the contribution of this paper, let us consider the example in Listing 1. The example[3] is written in Pharo Smalltalk.

```
1  ComposableModel subclass: #MethodBrowser
2      instanceVariableNames: 'listModel textModel toolbarModel'
3      classVariableNames: ''
4      category: 'Spec−Examples'
5
6  MethodBrowser>>#initializePresenter
7      listModel whenSelectedItemChanged: [:selection |
8          selection
9              ifNil: [
10                 textModel text: ''.
11                 textModel behavior: nil.
12                 toolbarModel method: nil ]
13             ifNotNil: [:m |
14                 textModel text: m sourceCode.
15                 textModel behavior: m methodClass.
16                 toolbarModel method: m ]].
17     self acceptBlock: [:t | self listModel selectedItem inspect ].
18     self wrapWith: [:item | item methodClass name,'>>#', item selector ].
```

Listing 1: The run-time type of the argument "**item**" cannot be statically detected by the traditional approach

Lines 1-4 define a class named MethodBrowser used to browse methods of a given class, and lines 6-18 define a method named initializePresenter to initialize variables of the MethodBrowser class. Suppose that a developer needs to know the type of the block argument item in the last line of the method initializePresenter, either to understand which method with selector *i.e.,* the name of the method,

[3] This code snippet is actual code from the SPEC system:
http://www.smalltalkhub.com/#!/~Pharo/Pharo60/packages/Spec-Examples

methodClass will be invoked, or to understand the behaviour of the method. The only available information is that this variable needs to understand messages with selectors methodClass and selector. Polymorphic selectors are frequently used in Smalltalk [21] which means that a large number of methods implement these selectors. In the Pharo image[4] we used for our experiments, there are 20 methods with selector methodClass and 67 methods with selector selector. A simple analysis, such as that offered by *RoelTyper*, which is used as the basic approach in the paper, uses the information about messages sent to the variable, and presents the developer with fourteen classes whose instances understand both messages. RoelTyper also uses the information about assignments to the variables, which is missing in this case. Hence, these fourteen classes are presented to the developer without any particular order. This number is even higher since a developer is presented only with the root classes that understand the corresponding set of messages. Even though it is obvious to the developer that the type of the variable item is related to some representation of the method, it is not obvious which method representation of many is used. Some of them are considered "polymorphic", indicating that instances of these classes can be interchangeable.[5]

It would be useful for a developer to be presented with a sorted list of classes that promotes classes likely to be correct to the top of the list. In this paper we argue that the information collected from inline caches from different software runs can be useful to order inferred types in dynamically-typed languages. For instance, if we use the information available from inline caches, we can observe that ten out of fourteen possible classes were used as variable type at some point in the Smalltalk code execution. Classes that were used the most are, in that order, CompiledMethod, RBMethodNode, Context and RGMethodDefinition. CompiledMethod class is present at the first spot, and this class represents the runtime type of the variable item.

## 3. Related work

### *3.1. Static type inference*

The pioneer in this field was Milner [3] who developed a type inference algorithm for ML, a functional programming language. The algorithm supports

[4]Pharo 6.0 version 60324

[5]Class comments for the class RGMethodDefinition state that this class "is polymorphic with CompiledMethod" class.

parametric polymorphism, but not subtyping. It infers the type of a variable using constraints based on the variable's usage. The algorithm is sound: if an ML program is well-typed, it will not produce run-time type errors. Another type inference algorithm is proposed for the functional language, FL [13]. FL is a dynamically-typed language, in which a type is considered to be a set of normal-form expressions.

Agesen *et al.* have developed a type inference algorithm [22] for *Self*, a dynamically-typed language inspired by Smalltalk [23], but based on cloning objects rather than instantiating classes, and supporting the possibility for dynamic and multiple inheritance. This is the first algorithm to consider dynamic and multiple inheritance.

The Cartesian Product Algorithm, known as CPA [4], exploits the fact that the type of the return value of a method usually depends on the types of the method's arguments. The algorithm itself has been developed for the *Self* programming language [24]. The authors use as a base algorithm the one described by Palsberg *et al.* [25] who modelled a program as a set of constraints.

Starkiller [6] is a type inference algorithm for Python, based on the Cartesian Product Algorithm. It tries to reconstruct a flow of objects to model the runtime behavior of a software. Each variable is represented as a node that holds the information about the possible types that a variable may assume at runtime.

CPA was used in combination with type hints from method argument names in Smalltalk [26]. The combination, named CPA* increased the size of the analysed call graph by about 30% and the number of method arguments for which it inferred the correct type by about 80%.

Spoon *et al.* developed one of the most precise type inference algorithms [7, 27], a demand-driven algorithm using subgoal pruning. The information used for type inferencing is analysed on demand, and the precision of the algorithm decreases if some of the subgoals required for type inference need to be discarded for complexity reasons.

A fast and relatively accurate type inference technique was developed by Pluquet [8]. The algorithm traverses the set of messages sent to a variable along with assignments to the same variable and infers types for it. This approach was used as a basis for our prototype implementation, as well as for the prototype implementation of the EATI algorithm, which advocates the idea of using the information available in the language ecosystem to increase precision [28].

### 3.2. Dynamic type inference

One of the first uses of inline caching as a way to statically reconstruct the type of a variable from a running system has been in Self [14]. The authors collect run-time receiver types seen at each call site, and feed this information back to the compiler for optimisation purposes. They predict the type of the receiver based on the receiver's types seen during previous program runs.

Static type inference that relies on dynamic collection of data has been developed for Smalltalk [5]. The algorithm observes types of objects stored in the variables at run time, and incrementally updates the static type information. The main assumption needed for this work is that test coverage is "complete" and that the program of interest is in a runnable state. Obviously, the more frequently the variable has been encountered during a program run, the more precise will be the type information.

A type inference algorithm for Ruby based on source code instrumentation and dynamic execution of the program has been developed by An *et al.* [10]. This approach needs to run the program for which the types of variables have to be inferred, and in order for the analysis to be sound, the execution must cover all possible paths in the control-flow graph. Type constraints are created for a variable based on the observed run-time types. As the authors state, the current overhead introduced with this analysis is quite high.

Odgaard *et al.* present a way of annotating JavaScript code based on the dynamic type information collected from running unit tests [12]. Code is instrumented in order to record run-time types of the variables. Instrumentation adds to software size and increases the execution time.

### 3.3. Hybrid analysis

The combination of static and dynamic analysis is known as *hybrid analysis* [29]. The key idea is to infer conservative information by use of static analysis, which is further fine-tuned by information collected at run-time. Fast and precise hybrid type inference has been presented for JavaScript [30], which tries to infer sound type information by customising static type information to also deal with types recorded at run time. These approaches are useful in cases where static analysis tends to produce too conservative data, or to be unsound due to dynamic class loading or reflection.

### 3.4. Gradual typing

Gradual typing allows some of the variables to be typed at compile time, while other variables may be left untyped. Correctness of typed variables is checked at

compile time, and the type system ensures that these types are not violated at run time. Gradual typing has been implemented for Racket, a multi-paradigm programming language [31]. It supports statically-typed Racket programs by supporting existing language idioms. It has been used as an inspiration for *Gradualtalk*, a gradual type system for Smalltalk developed by Allende *et al.* [11].

A gradual type system called *DRuby* [32] has been implemented for Ruby, allowing developers to annotate and type check selected parts of the code base.

### *3.5. Optional typing*

Optional typing grants to a developer the opportunity to specify the type of a variable when deemed necessary. Strongtalk is an optional type system developed for Smalltalk [33], which was used for optimisation purposes. Pegon [34] is an optional type system also developed for Smalltalk and inspired by Strongtalk. A pluggable type system allows a language to support multiple, optional type systems.

## 4. Gathering of dynamic type information

### *4.1. Dynamic type information gathering*

Object-oriented languages are very difficult to optimise statically due to the dynamic nature of message sends: the method called is unknown at compilation time. To solve this problem, modern virtual machines rely on a JIT (Just-in-Time compiler) which speculates on types based on the types met in the previous runs of the code to achieve high performance. We built a dynamic type data gatherer using the same infrastructure to extract types from previous runs of the JIT.

We describe shortly the execution model used by such VMs, especially the techniques used to extract type information from previous runs. We then explain how we use the infrastructure provided to gather type information.

### *4.2. High-performance virtual machine execution model*

There are two main ways to achieve high-performance in object programming languages in the presence of a large number of message sends.

Part of the community believes that the highest performance can be reached using a meta-tracing runtime compiler. Such compilers record a linear sequence of frequently executed operations and translate them efficiently to machine code. The most popular open-source production VMs using this design are LuaJIT [35]

and Pypy[6] [36]. The Pharo VM does *not* use this design, so we will not discuss it further. It is not clear whether our dynamic type data gatherer could work with such VMs as the execution model is quite different.

The rest of the community believes that the highest performance can be reached using a method-based runtime compiler. Such compilers optimise frequently-used methods by using type information of previous runs. The most popular open-source production VMs using this design are Java's hotspot VM[7] [37] and Javascript's V8 engine[8] [38]. The Pharo VM implementation is heading toward this design, though speculative optimisations have not been introduced yet. For the rest of this section we will discuss only this design.

### *4.3. Execution of message sends*

*Interpreter.* Conceptually, the VM executes compiled methods by interpreting the method's bytecodes. Each message send causes the bytecode interpreter to attempt to fetch the method to execute from the receiver class and the message selector in a global look-up cache, and on a cache miss, it performs the look-up routine. As the interpreter's cache is global, it is difficult to provide reliable type information from interpreted code[9].

*Baseline JIT.* When a method is used multiple times, the baseline JIT (in contrast to an optimising JIT) translates it to machine code and the VM subsequently uses that version. The machine code version of the compiled method produced by the baseline JIT includes a specific cache for each message send, called an inline cache [39, 40]. Such caches remember look-up results for a given receiver type, speeding up the execution and, as discussed later, providing type information for the optimising JIT.

An inline cache can be in four main states. If one stops the execution of a typical production application and analyses the inline caches present in the machine code generated by the JIT, the inline caches would be present with the following frequency (please, bear in mind that this is an approximation based on the analysis of real production applications to provide orders of magnitude):

[6] Pypy is mostly used as an alternative virtual machine for Python.

[7] The hotspot VM is the default virtual machine for Java.

[8] V8 is mostly used as the Javascript engine for Google Chrome and NodeJS.

[9] It is possible, but it requires either significant overhead or to use processor-specific machine code optimisation techniques, implying in this latter case a different version of the code for each processor supported.

- **unused:** 30% of inline caches remain unused, for example because they are in a branch that is never reached by the running code. All inline caches are compiled to unused inline caches, and they become monomorphic when executed for the first time.

- **monomorphic**: 63% of inline caches (90% of used inline caches) contain a single receiver type. In this case the cache is entirely present in the message send's method machine code. This cache becomes polymorphic when another receiver type is met.

- **polymorphic:** 6.3% of inline caches (9% of used inline caches) contain a small number of different receiver types (up to 4 in Oracle's Java virtual machine, 6 in the Pharo VM). In this case the cache is extended to a remote jump table, generated in machine code. This case becomes megamorphic when too many receiver types are met.

- **megamorphic:** 0.7% of inline caches (1% of used inline caches) are used with more than a small number of different receiver types. This latter case can be implemented in multiple ways, but most in popular implementations there is no reliable way to extract types from this kind of cache.

When introspecting the machine code of a method, the VM can read the inline cache's data. The VM starts by identifying the state of the cache and inferring types accordingly. If the cache is unused or megamorphic, no type can be inferred. If the cache is monomorphic or polymorphic, the type of the receiver can be inferred as a small set of one to six concrete types.

We note that the caches are not guaranteed to contain all the types met for a given send site. Not all of the possible executions of the send site by the bytecode interpreter are recorded. In addition, the inline cache cannot reference methods not already compiled to machine code, hence if the cache misses and the method found is interpreted, it is not recorded in the cache. The cache data is normally used as a hint to direct compiler optimisation and is good enough for this purpose, but it is not 100% accurate.

*Optimising JIT.* Once a method is executed numerous times, the optimising JIT extracts the type information through machine code introspection and generates a new machine code version of the method that is faster to execute. Every optimising JIT performs inlining based on the types speculated, deoptimising the code on-the-fly if one of the compilation-time speculations happens to be incorrect in

subsequent runs, so the execution of a message send at this point does not really make sense any more (there is no call at all after inlining).

### *4.4. Dynamic type information gatherer built*

As stated before, the Pharo VM has in production a bytecode interpreter and a baseline JIT, but the optimising JIT is still a work in progress. The direction taken for this work is similar to the Truffle project in the sense that the optimising compiler is written in Pharo itself and runs in the same runtime as the optimised application. Because of these specifications, it is possible from Pharo to ask the VM about the inline cache's values so the optimising JIT can speculate about types. This was done through a new primitive method[10] answering the inline caches values for a given method if the method has a machine code version generated by the baseline JIT. This part of the work is fully implemented, tested and running, hence it is possible to use it for other applications, such as the dynamic type information gatherer.

For each send site, the type information provided by the VM consists of an array of types and methods next to the bytecode program counter of the send site. As types are inferred at the AST level in our other heuristics [15], we needed to leverage that information. The Opal compiler [41] provides a tool to map the bytecode program counter to AST nodes, which is normally used by the debugger [42] to highlight the code being executed. We used this tool to map the type information from the bytecode program counter to the AST nodes.

The VM can provide type information only on methods compiled to machine code. We added another primitive method, answering an array of all methods compiled to machine code, *i.e.,* all methods with type information. This way, it is possible to get the list of methods with runtime type information and collect the types of each of those methods for each message send's receiver.

Our dynamic type data gatherer is scheduled to run regularly (every second) in the Pharo image,[11] in order to ensure that collected data is up-to-date. The gatherer queries the virtual machine for all the message sends that have recently

[10] Some messages in the system are responded to primitively. A primitive response is performed directly by the interpreter rather than by evaluating expressions in the method. Essential primitives, as opposed to optional primitives available for performance only, cannot be performed in any other way. For example, Smalltalk without primitives can move values from one variable to another, but cannot add two SmallIntegers together.

[11] The term "Pharo image" is used to denote snapshot of the running Pharo system, frozen in time.

been executed, and collects type information of the receiver.

### *4.5. Type information and optimising JIT*

The current version of the dynamic type data gatherer extracts type information from the machine code versions of methods generated by the baseline JIT. The production Pharo VM features only a bytecode interpreter and a baseline JIT, hence the type information is extracted easily. However, VMs such as the Java virtual machine feature in addition an optimising JIT compiler. Introspection of the machine code generated by the optimising JIT of such VMs is very difficult because it is heavily optimised and there is close to no type information available due to optimisations such as speculative inlining.

The Pharo VM is heading in the same direction as an optimising JIT is under development. At this point, we will need another way to extract types in optimised code. We believe that when the optimising JIT extracts the type information from previous runs of multiple methods (*i.e.,* the method it optimises as well as the methods it is inlined in) to speculate on types, it could inform external tools such as the dynamic type gatherer, about the read types. Therefore, it should be possible to implement a similar tool on the incoming releases of the Pharo VM as well as on the Java virtual machine.

## 5. Type inference algorithm

### *5.1. Terminology*

To explain ICTI, we introduce a simple set-theoretic model in Figure 1 that captures key properties for the entities shown in the UML diagram in the Figure 2.

$$msg : V \rightarrow \mathcal{P}(S) \tag{1}$$

$$sel : M \rightarrow S \tag{2}$$

$$def : M \rightarrow C \tag{3}$$

$$sup : C \rightarrow C \cup \{null\} \tag{4}$$

$$assign\_types : V \rightarrow \mathcal{P}(C) \tag{5}$$

$$under : C \cup \{null\} \times S \rightarrow \{true, false\} \tag{6}$$

Figure 1: The core model.

Given a target programming language, C is the domain of all classes, M is the domain of all methods, S is the domain of all selectors. V is the domain of all variables, including instance variables, method arguments and local variables.

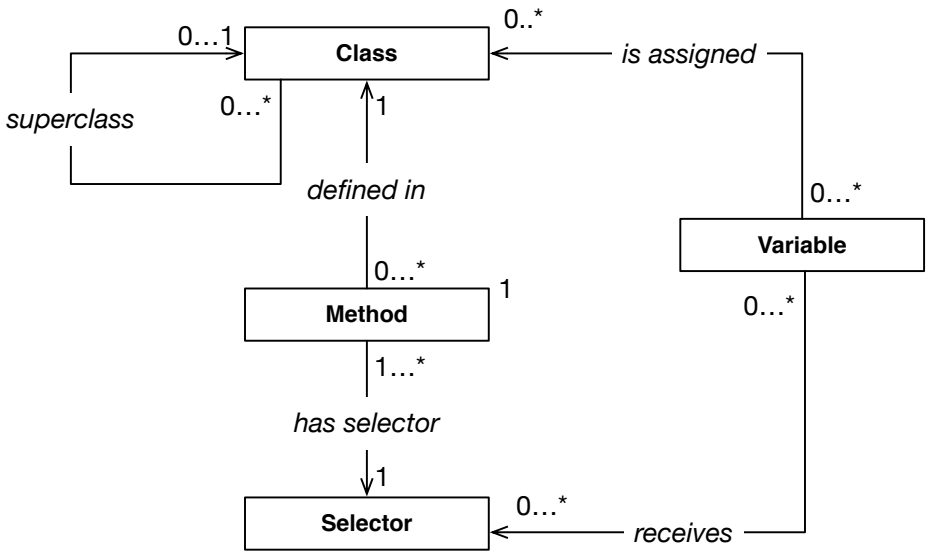


Figure 2: The core model in UML

Each variable $v$ has a (possibly empty) set of messages $msg(v)$ sent to it within its lexical scope[12] either directly or through the getter methods (1). We call this set of messages the *interface* of the variable $v$. Each method $m$ has a unique selector $s = sel(m)$ (2), and is defined in a unique class $c = def(m)$ (3). Each class $c$ has a unique superclass $c' = sup(c)$(4). We define the superclass of Object to be *null*, *i.e., sup*(Object) = *null*.

Consider the example class hierarchy in Figure 4. In this example we see a class HandMorph with an instance variable named temporaryCursor and methods cursorBounds, showTemporaryCursor:hotSpotOffset: and initialize. Within these three methods messages sent to the instance variable temporaryCursor are

$$msg(\text{temporaryCursor}) = \{\text{extent, offset, ifNil:ifNotNil:}\}$$

[12]Note that in this paper we consider the scope for instance variables only to be the methods of the class in which it is defined, but not its subclasses. Considering also the methods of the subclasses to be part of the lexical scope of a variable can only improve the results.

$$\textit{under}(c, s) = s \in \textit{sel}(\textit{def}^{-1}(c)) \vee \textit{under}(\textit{sup}(c), s) \tag{7}$$

$$\textit{intr}(c) = \{s \in S | \textit{under}(c, s) = true\} \tag{8}$$

$$\textit{sel\_all\_types}(v) = \{c \in C | \textit{msg}(v) \subseteq \textit{intr}(c)\} \tag{9}$$

$$\textit{roots}(C') = \{c \in C | \forall n > 0, \textit{sup}^n(c) \notin C'\}, \text{where } C' \in \mathcal{P}(C) \tag{10}$$

$$\textit{sel\_types}(v) = \textit{roots}(\textit{sel\_all\_types}(v)) \tag{11}$$

Figure 3: Computing possible types for a variable.

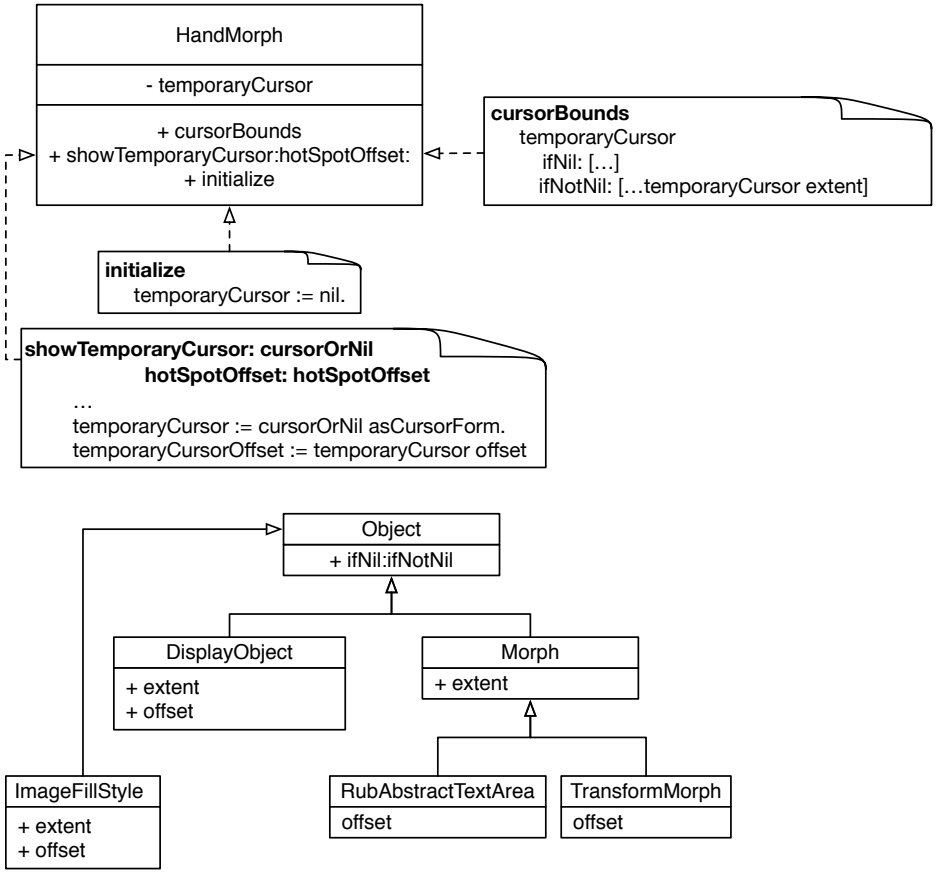


Figure 4: Sample class hierarchy

Also, each variable $v$ may have one or more assigned types $c \in \textit{assign\_types}(v)$

| Expression | Inferred type |
|---|---|
| $x = y$ $x == y$ $x < y$ $x > y$ $x <= y$ $x >= y$ $x \ = y$ | Boolean |
| $x\ msg\ y$, where $msg$ is any of the arithmetic, logarithmic or trigonometric functions or functions used to round a number | Number |

Table 1: Heuristics from RoelTyper [8] used to infer the type of the expression

if the variable $v$ occurs on the left-hand side of an assignment where the right side of the same assignment is a message send to a class resulting in the creation of a new object, *i.e.,* is a call to a constructor, or this newly-created object has been assigned to the variable via setter methods (5). Multiple assignments to the same variable are also possible.

We have used a couple of heuristics to guess the type of the expression result assigned to the variable, as was done with RoelTyper [8]. These heuristics are listed in Table 1.

We can now query the model to ascertain the set of possible types for every variable. Each class $c$ can either understand the selector $s$ or not (6). The class $c$ understands selector $s$ if it defines a method $m \in \mathit{def}^{-1}(c)$ such that $\mathit{sel}(m) = s$ or its superclass $\mathit{sup}(c)$ understands it (7), as presented in Figure 3. We also define that *under*(*null*, s) = *false*. The interface of the class $c$ is a set $\mathit{intr}(c)$ of all the selectors $s$ that class $c$ understands (8). We first create set of all classes which understand the interface of the variable $v$ (9), and then take the roots of hierarchies of this set of classes (10) as possible types for a variable $v$ (11).

In the example in Figure 4 we see that the root classes that understand the set of messages sent to variable temporaryCursor are classes DisplayObject, ImageFillStyle, RubAbstractTextArea and TransformMorph, *i.e.,*

*sel_types*(temporaryCursor) = {DisplayObject, ImageFillStyle, RubAbstractTextArea, TransformMorph}

It is important to emphasise that our aim is not to provide receiver type information directly from the inline caches, if available, to a developer. This information, available or not at the moment of inferring variable's type, may not be comprehensive. During the lifetime of the image, we collect the information about the frequency of use of each class as a type of the receiver in the current image. Possible classes for a receiver type, which are statically inferred using

the message sends and assignments to the variable, are then sorted based on this frequency.

### *5.1.1. Method arguments*

A common practice in dynamically-typed languages is to name method arguments (*i.e.,* formal parameters to methods) in such a way as to reveal their expected type [43, 44, 45]. Type hints in a variable's name have a positive impact on program comprehension [2]. Type annotations in method argument names are used in several dynamically-typed languages like Python [46], Groovy [47], Dart [48] and Smalltalk [49]. Smalltalk developers insert the name of the expected type into a method argument name [23, 43]. For example, a method argument named aString suggests that the method expects an argument of type String. Recent studies revealed that developers do not practice this pattern consistently [49], but it is used as a heuristic. We have implemented the following heuristic that restricts the set of types inferred for method arguments:

1. first we extract the substring of the argument name starting from the first uppercase character to the end, *e.g.,* a variable named aBlock would yield Block

2. among classes that are inferred from a variable's interface, we would select only those classes whose name matches the regular expression ".*",*extracted substring*,".*"

We apply this heuristic only to the set of types inferred from variable interface, that is *sel_types*, as we deem assigned types to the variable to be correct as it is. However, this is beyond our small example.

### *5.2. Dynamic information*

Let *MS* be the set of all message sends in the target programming language. Each message send has a receiver and a selector sent to the receiver.

Each class occurs as the type of a receiver for a message sent zero or more times (12). Based on the inline cache information collected during the image lifetime, we calculate the *class_freq* (13), as the number of message sends for which this class occurred as a receiver type during run time. *class_freq* is a global variable calculated for each class. From this information we calculate *class_value*$(c)$ (14) for class $c$, as the sum of *class_freq*$(c')$ for each class $c'$ which is a subclass of $c$. This information is used to sort the classes that represent possible types for

$$\textit{run-time\_type} : \textit{MS} \rightarrow \mathcal{P}(C) \tag{12}$$

$$\textit{class\_freq}(c) = |\{ms | c \in \textit{run-time\_type}(ms)\}| \tag{13}$$

$$\textit{class\_value}(c) = \sum_{\substack{c=sup^n(c'), n \in N \\ c' \in C}} \textit{class\_freq}(c') \tag{14}$$

Figure 5: Calculating class value.

a variable. We extract this information from the virtual machine, with the help of the implemented dynamic type information gatherer.

We convey here the *class_values* for each of the classes present in the set *sel_types*(temporaryCursor), as collected during dynamic type information gatherer phase. In the interest of conciseness, we convey only the *class_values* of these classes, and not the lists of message sends for which these classes were recorded as receiver types at run time. The details of the dynamic type information gatherer process will be explained in subsection 6.1.

Dynamically collected information is used to order separately two sets of classes: *assign_types*($v$) and *sel_types*($v$). In our example in Figure 4, we encounter the following values of the classes DisplayObject, RubAbstractTextArea, TransformMorph and ImageFillStyle :

$$\begin{aligned} \textit{class\_value}(\text{DisplayObject}) &= 396 \\ \textit{class\_value}(\text{RubAbstractTextArea}) &= 343 \\ \textit{class\_value}(\text{TransformMorph}) &= 244 \\ \textit{class\_value}(\text{ImageFillStyle}) &= 31 \end{aligned}$$

Based on the obtained information, we can now sort the possible types for the variable temporaryCursor. The list *assign_types*(temporaryCursor) in our example has no elements, so there is nothing to sort. But the list *sel_types*(temporaryCursor) has four elements that will be sorted as follows:

1. DisplayObject
2. RubAbstractTextArea
3. TransformMorph
4. ImageFillStyle

To present one list of possible types to a developer, we use *AssignmentFirstMerger* from the base approach [8]. This means that we give dominance to assignment types rather than selector types. After sorting both lists of types, namely *assign_types*(v) and *sel_types*(v), we iterate through the list of *sel_types*(v) and remove all the classes that are related to any of the classes from *assign_types*(v), *i.e.,* are a superclass or a subclass of any of the assignment types. We append the remainder of the sorted list of selector types to the list of assignment types.

### *5.3. Class-Based approach*

While it is important for developers to know the possible hierarchy to which the type of a variable may belong, it is sometimes also important to infer the precise class that represents the run-time type of the variable. Many of the analysed variables have an interface understood by many independent hierarchies, *i.e.,* hierarchies whose roots do not have a common superclass understanding the same interface, thus we wanted to verify how successfully ICTI would infer the precise class. A variable can have an interface understood by tens, hundreds, or even thousands of classes. Obviously, such information presented to a developer is not helpful. Hence we order them so that we promote the correct class towards the top of the list.

Most of the existing type inference algorithms for dynamically-typed languages focus on the type hierarchy rather than on the precise type.

The hierarchy-based approach has already been explained throughout subsection 5.1. The class-based approach takes into account all the possible classes inferred based on the variable's interface. This means that sorting classes is now applied to *sel_all_types*($v$) rather than to *sel_types*($v$). This indicates that *class_value*($c$) = *class_freq*($c$) for any class $c$, since we are considering each class separately as the possible type. Merging with the list of *assign_types*($v$) is performed in the same manner as in the hierarchy-based approach.

In the example in Figure 4 the interface of the instance variable temporaryCursor is understood by four independent hierarchies of classes. If we include as possible types also their subclasses, we get a set of 20 different classes, which we do not list here.

Accordingly, we present two types of information to the developer: the one obtained by a hierarchy-based approach, and the other by a class-based one.

Let us emphasise here that we do not apply this change to the set *assign_types*($v$), but only to the set *sel_types*($v$). We consider the set of explicitly assigned types to a variable to be truthful, as it is. The implications of this decision are discussed in section 7.

## 6. Evaluation

### *6.1. Inline caching type gathering*

In previous work type information from inline caches was fed back to the compiler for the purpose of optimisation. To the best of our knowledge, this is the first experiment that explores to what extent run-time type information from other packages is useful when trying to statically infer types in the packages separate from those whose run-time types are collected.

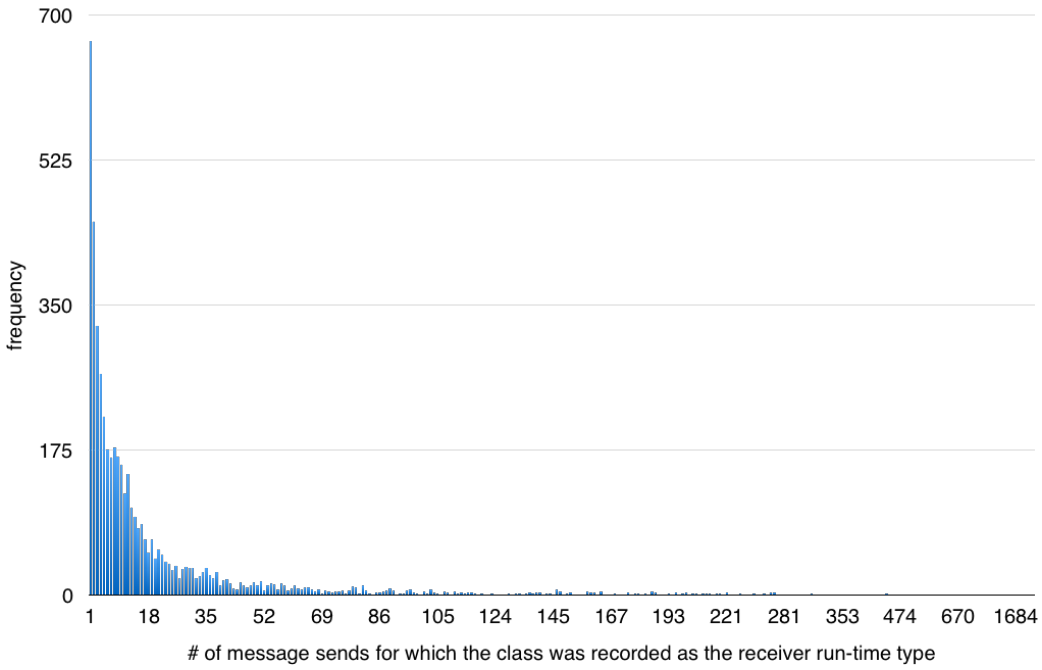


Figure 6: Distribution of *class_values* for recorder run-time types

In order to collect run-time type information from inline caching we have run almost all the tests available in the Pharo image.[13] We ran 815 test classes, which left us with almost 12 000 test methods. These tests allowed us to collect the frequency of classes as message receiver types at run time for 4925 out of 7640 classes available in the image, that is about 65% of classes. The collected data has been used to calculate the class value, $class\_value(c)$ of each class $c$. We

[13] Pharo 5.0 version 50761

have run the Shapiro-Wilk test [50] on the set of run-time types collected during tests execution and obtained the *p-value* to be less than $2.2e$-16 which means that the run-time types are not normally distributed. The distribution of *class_values* can be found in Figure 6. We have measured the test execution times during data collection and compared them to test execution without inline cache data collection. The average overhead introduced per method is 0.6 milliseconds, *i.e.,* a bit less then 40%. The introduced overhead is acceptable, even though it is high, since we have run an unoptimised version of the type gatherer. We have run all the tests cases at once, and instructed the type gatherer to run every second, to collect enough of data from inline caches. We believe that in reality the type gatherer can be instructed to run more seldom, thus the introduced overhead would be much smaller.

We are fully aware that the distribution of *class_values* for the classes recorded from inline caches during the test executions influences the results to a great extent. The information collected from the inline caches, even though is correct, may not be complete. This problem arises from the limited amount of memory that the virtual machine uses to store the information from the inline caches. For the purpose of the experiment, we have run all test cases in the image sequentially. Because of that, we have instructed the dynamic type gatherer to run every second, so that we lose as little run-time information as possible. We have discussed these limitations in Section 7.

### *6.2. Projects used for evaluation*

For the evaluation we have used four open-source Pharo projects for which we were able to collect run-time information that closely depicts their real usage: Glamour[14] [51], Roassal2[15] [52], Morphic [53] and Moose[16] [54, 55, 56]. Glamour is a framework for specifying the navigation flow of browsers. Roassal is an agile visualisation engine that graphically renders objects. Morphic is a User Interface construction kit, and Moose is a platform for software and data analysis.

We first statically infer types for the variables defined in these projects, and proceed by ordering these types based on the class usage frequency collected from inline caches. On the other hand, we collect run-time types of these variables, through the execution of *example methods*. These methods are defined by

[14] http://www.smalltalkhub.com/#!/~Moose/Glamour
[15] http://smalltalkhub.com/#!/~ObjectProfile/Roassal2
[16] http://www.smalltalkhub.com/#!/~Moose/Moose

developers of these projects, and intended as the examples of how the corresponding software should be used. Glamour has 83 of these methods, Morphic 29 and Roassal 952. For Moose we have collected run-time data by performing software analysis on a project. We have loaded an *MSE* file,[17] and, after loading the model, we performed the following analysis of the project: we have searched for all deprecated classes and deprecated methods. We have also searched for largest methods in the model, as well as for the abstract methods and methods that contain a null check. During the execution of these projects, the information about types of variables was recorded, and this information was declared to be the *ground truth*. In order to recover these run-time types of the variables, the source code of the projects was instrumented to log the types of the variables as the provided examples were executed. To recover dynamic type information we have used a tool to track the types of variables at run time, built on top of Reflectivity,[18] a reflection API for annotating AST nodes with metalinks. For each variable in these five projects, we have compared the list of types statically inferred by ICTI with the list of types recorded at run time, during the execution of example methods.

Looking back at the inline cache type gathering phase, we run 815 tests that are part of the standard Pharo image. Note that two of the projects we have used for the evaluation, namely Glamour and Morphic, are part of the default Pharo image. During the inline cache type gathering phase we have omitted tests that belong to these projects. After removing the 76 tests belonging to these two projects from the 891 tests in the Pharo image, we were left with 815 tests.

We use run-time type information collected by running example methods to represent the ground truth in the evaluation phase, and compare them to statically inferred types. We do not collect any type information from these four projects during the inline cache type gathering phase.

Types of these variables are then inferred using ICTI. Examples that we used to collect the run-time information about types for which we were able to statically infer the type, *i.e.,* at least one message was sent to the variable, or there was an assignment of a newly instantiated type to the variable, covered 179 variables in Glamour, 257 variables in Moose, 1052 variables in Morphic and 3998 variables in Roassal2.

| Hierarchy-based approach | | | | | | | |
|---|---|---|---|---|---|---|---|
| Project name | #of analysed variables | #of guessed variables | #of guessed variables - library type | #of guessed variables - package type | #of near-guessed variables | #of in-correctly-guessed variables | #of Object type |
| Roassal | 3998 | 2401 | 1411 | 987 | 513 | 776 | 308 |
| Glamour | 179 | 108 | 50 | 57 | 24 | 14 | 33 |
| Morphic | 1052 | 705 | 442 | 263 | 175 | 66 | 106 |
| Moose | 257 | 158 | 144 | 14 | 34 | 50 | 15 |
| SUM | 5486 | 3372 (61.5%) | 2047 (61%) | 1321 (39%) | 746 (13.6%) | 906 (16.5%) | 462 (7.9%) |

Table 2: Inline cache heuristic

### *6.3. Overall results — Hierarchy-Based approach*

If the class at the top of the list of sorted types is a superclass of the class which represents actual run-time type of a variable, we label such a variable *guessed*. Around 20% of variables used for the evaluation were recorded to have more than one run-time type. In this case, we applied the following criterion. Let us denote with $n > 1$ the number of recorded run-time types. We define $m \geq 1$ to be the size of the smallest set of statically inferred classes whose combined subhierarchies include all run-time types of the variable, and label this set with $CT$ (*correct types*). If $CT$ set is equal to the set of first $m$ classes of the statically inferred list, we consider the variable to be guessed.

Overall results are presented in Table 2. Results are presented per package, as well as in total. The results show that the heuristic of ordering types based on the frequency of a type being seen as the run-time type is able to *guess* type in around 61.5% of cases. Deeper analysis revealed that, expectedly, ICTI improved type inference of library types more than type inference of project-related types.

If ICTI failed to guess the type of a variable, but the correct type is present the top three classes, we call such a variable *near-guessed*. If the variable has $n$ run-time types, where $n > 1$, we consider it as near-guessed if the set of first $m + 2$

[17] MSE is an exchange format for Moose models, analogous to XML

[18] http://www.smalltalkhub.com/#!/~RMoD/Reflectivity

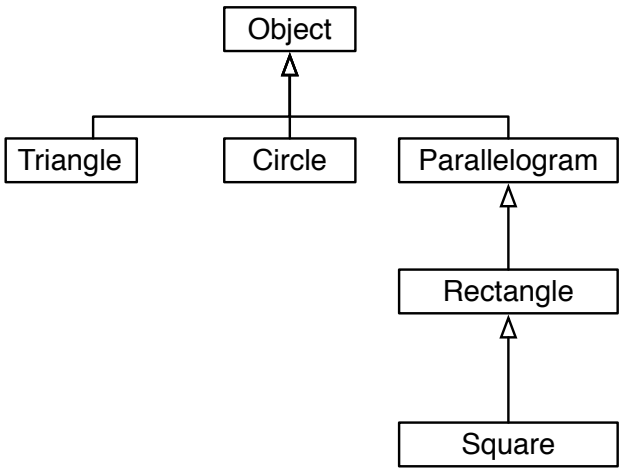


Figure 7: Sample class hierarchy

types of the statically inferred list of types is a superset of $CT$. That is we allow ourselves to make two mistakes, and miss the correct types by two spots. For example, let us imagine a hierarchy of classes presented in Figure 7. Let us also imagine that a variable has Rectangle and Square as run-time types, and statically-inferred list of types included in the first three positions, respectively, Circle, Triangle and Parallelogram. Both run-time types are contained in the subhierarchy of the Parallelogram class, thus $m = 1$. The set of the first $m$ statically-inferred classes, *i.e.,* the Circle class, does not include in its subhierarchy all the run-time types of the variable, so the variable cannot be labeled as guessed. Combined subhierarchies of the classes Circle and Triangle do not include all run-time types. But, the combined subhierarchies of the classes Circle, Triangle and Parallelogram do include all run-time types, thus the variable is considered to be near-guessed. In this case we miss the correct type by two. On the other hand, if the variable has Rectangle and Square as run-time types, and statically-inferred list of types includes in the first four spots, respectively, Circle, Triangle, Square and Rectangle we also consider it to be near-guessed. In this case $m = 2$ as two statically-inferred classes, *i.e.,*Square and Rectangle are needed to cover all run-time types. We have also missed the correct types by two, as the classes Circle and Triangle do not include in their subhierarchies either of the recorded types.

ICTI was additionally able to *near-guess* the type for about 13.6% of variables, thus making the results optimal for 75% of variables in total.

In all other situations, that is, if the variables is neither guessed nor near-guessed, the variable is labeled as *incorrectly-guessed*.

As for the remaining set, we find 462 variables for which we were not able to infer any other type than Object. This means that no assignment was performed to the variable, nor was any message sent other than messages defined in Object class. We argue that these results could be discarded since they are easily identifiable and they do not provide any useful information to the developer. Aside from 441 methods already defined in the Object class in Pharo, it is also possible to add user-defined methods in library classes. For example, the selector rtValue: is specific to the Roassal2 package, but it is implemented in the Object class, and thus can be sent to any object in the image. This leaves us with 906 variables that are incorrectly-guessed.

To be able to assess the improvement of ICTI compared to the basic algorithm, we have compared these results with the scenarios in which the basic algorithm infers nothing but the correct type of a variable. This comparison also allows us to evaluate the differences between ICTI and an existing type inference algorithm, *i.e.,* RoelTyper [8]. The comparison is presented in Table 3. These results show that ICTI inferred correct types as the first ones on the list of possible types for significantly more variables than the basic algorithm *i.e.,* 132% improvement. The idea behind ICTI is to promote the correct type to the top of the list, and reduce the number of classes in the list that are present on the list before the correct type of a variable. In its current state, the basic algorithm succeeds to infer unambiguous results for a bit more than a quarter of all variables. For the remaining of variables, the provided list of possible classes contains in some cases more than a hundred inferred types. Such an unordered list may not be useful to a developer. The aim of ICTI is to promote the correct type to the top of the list, and decrease the number of classes that are falsely ordered before the correct one.

One may argue that the obtained precision of ICTI may not seem to be very high. However, we aimed for a way to improve the precision of a simple type inference algorithm, regardless of its intended purpose. ICTI, as one of these ways, was able to double the number of variables for which it inferred correct type information when compared with the underlying approach. For that reason, we deem the obtained trade-off between precision and speed as acceptable.

### *6.4. Difference in between basic algorithm and ICTI*

The basic algorithm does not provide any ordering of the possible types of variables but presents them in random order. When it infers as possible types a set of classes that is a superset of the correct variable types, this information may lead

| Correctly inferred types | |
|---|---|
| Basic algorithm | ICTI |
| 1453 | 3372 |

Table 3: Comparison with basic algorithm

to wasted developer effort and cause more harm than good. Promoting the correct type to the top of the list can be quite challenging without any flow analysis. We consequently investigate how large are the lists of possible types for variables for which the basic algorithm was not able to infer the correct type, and which were guessed by ICTI, *i.e.,* for 1919 variables. In Figure 8 we can see around 19% of variables, *i.e.,* 377 variables out of 1919 have two statically inferred types. For the remaining majority, the number of possible types is 3 or larger. If we remember that these types represent only hierarchy roots, the importance of promoting the right type to the top of the list is even higher. The median length of these lists is seven, and the maximum is 1882, indicating that ICTI was able to promote the correct type to the top of the list even for lists larger than a hundred.

### *6.5. Not guessed variables*

We provide a deeper analysis of the reasons why for 906, *i.e.,* 16.5% of variables ICTI was not able to correctly infer types.

#### *6.5.1. Run-time types are not inferable without flow-sensitive analysis*

A bit less than 6% of incorrectly guessed variables, *i.e.,* 54 have an interface which is not understood by any of their run-time types. Nine of these variables have an interface which is not understood by any class in the image. After closer investigation, we have discovered that these variables in most cases have an interface containing type predicates, *i.e.,* message sends that ask the variable for its type and program execution continues based on the answer. One of the most frequently occurring examples is presented in the Listing 2 where the temporary variable myselfRescoped is queried for its type, and the execution flow is then divided according to whether it is a Collection or not. In this example, the type predicate isCollection is implemented in two classes: Collection, in which the method simply returns true, and Object, in which it returns false. This is the most common scenario, which indicates that these messages can be understood by any class.

```
FAMIXNamedEntity>>moosechefEqualsTo: anEntity moduloScope: aScope
    | entityRescoped myselfRescoped |
```

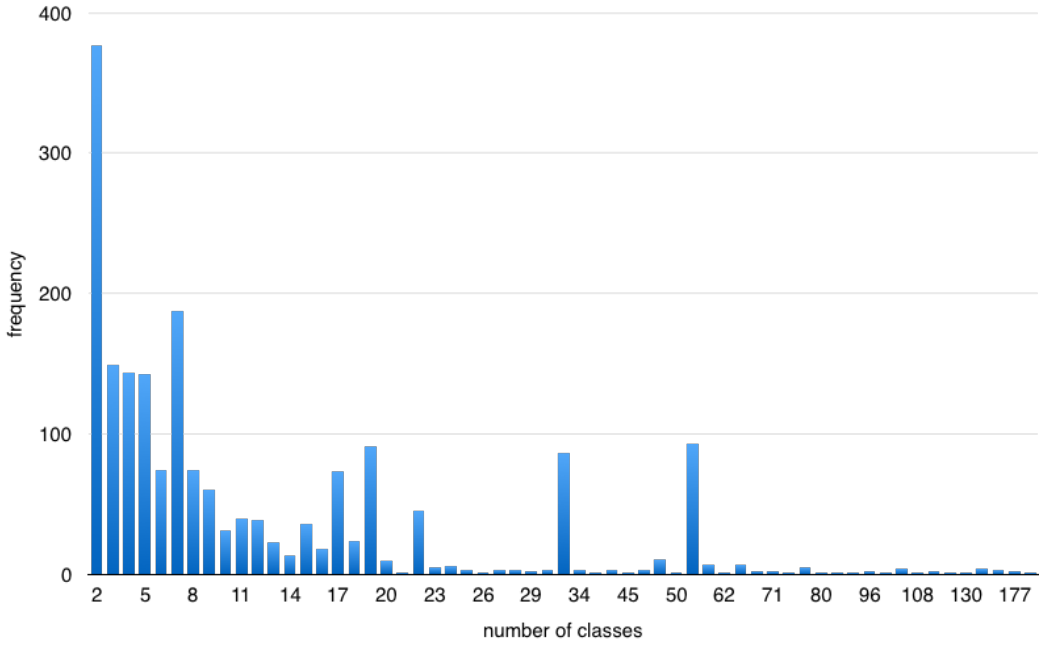


Figure 8: Statically inferred type sizes frequency

```
...
myselfRescoped isCollection
    ifTrue: [...]
    ifFalse: [...]
...
```

Listing 2: Type predicate isCollection usage example

Recent analysis [57] revealed that type predicates are used in almost all analysed projects to perform explicit type dispatch. Every 50th line of code contains a type predicate. However, the analysis of type predicates is not performed by the simple approach we have used in the basic algorithm.

### *6.5.2. Known duck-typed combinations*

*Duck-typing* [58] refers to the usage of variables to refer to objects of distinct classes which understand the same set of messages, without a common superclass understanding the same set. A classical example of duck-typing in Smalltalk is interchangeable usage of Symbols and BlockClosures, since instances of both

classes understand the message value:. Beside Symbols and BlockClosures, the most commonly occurring pairs of classes are Array and OrderedCollection, and CompiledMethod and ReflectiveMethod. We have found 38 variables used to point to both of types included in a duck-typed pair. These variables demand flow-sensitive algorithms for their precise inference. Otherwise, simple type inference algorithm may be improved for a specific language by including common duck-typed pairs into the analysis.

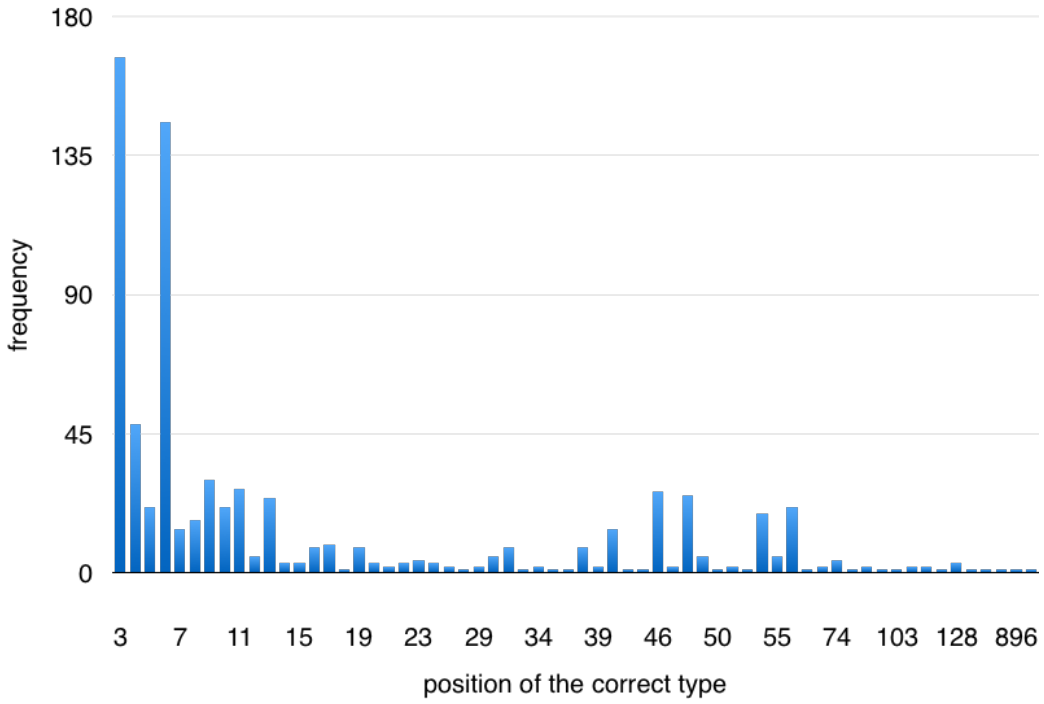


Figure 9: Position of correct type in the list

### *6.6. Position of the correct type*

We have investigated what is the distribution of $k$ where $k$ is the position of the correct type in the list of possible types. If we omit variables for which it is not possible to infer any other type than Object, the distribution of $k$ reveals that in 88% of cases the correct type was present in the first three spots of the list. For 3372 variables the correct type was present at the top of the list, while for 575 variables, the correct type was in second place. We present the distribution

| Class-based approach | | | | | | | |
|---|---|---|---|---|---|---|---|
| Project name | #of analysed variables | #of guessed variables | #of guessed variables, library type | #of guessed variables, package type | #of near-guessed variables | #of in-correctly-guessed variables | #of Object type |
| Roassal | 3'998 | 1534 | 586 | 947 | 314 | 1842 | 308 |
| Glamour | 179 | 74 | 35 | 39 | 17 | 55 | 33 |
| Morphic | 1052 | 351 | 258 | 92 | 107 | 488 | 106 |
| Moose | 257 | 100 | 97 | 3 | 29 | 113 | 15 |
| SUM | 5486 | 2059 (37.5%) | 976 (47.4%) | 1081 (52.5%) | 468 (8.5%) | 2498 (45.5%) | 462 (8.4%) |

Table 4: The inline cache heuristic

of positions three and onward in Figure 9. For more than 90% of variables the correct type is present within the first six inferred types.

### *6.7. Overall results — Class-Based approach*

Results of the class-based approach are presented in Table 4. As expected, control- and data-flow insensitivity of the algorithm took their toll. ICTI was able to correctly infer the precise type of the variable in 37% of cases, and to near-guess it for only 8.5% more variables. To the best of our knowledge, there is one other simple approach that tries to infer the precise type for a variable [15], and it shows very similar results.

## 7. Threats to validity

*Construct validity.* In order to implement ICTI for a dynamically-typed language, it suffices to be able to query the structure of the code to create a list of possible types for the variable, and to collect the type information from inline caches.

There are a couple of drawbacks when it comes to the virtual machine.

We are not always able to access all the types encountered at run time. In practice, the Cog virtual machine has an executable *machine code zone*, containing all the machine code versions of frequently used methods. Most frequently used methods and types are always present in the machine code zone with inline cache information. However, uncommon types may not be present. Indeed, if a

method was executed a single time, it may have been interpreted and hence not provide any type, since only methods executed multiple times are eligible to be present in the machine code zone. In addition, the machine code zone has a fixed size. Hence, when the zone is full, the garbage collector frees one quarter of the machine code zone, starting from the least used to the most used methods, losing all the type information present. To partially solve the garbage collection problem, we doubled the size of the machine code zone for our experiments.

*Internal validity.* The main threat to validity comes from the collected run-time information from inline caches. If a class represents a variable's run-time type, but was not used during previous system runs, it would be missed, and results will not be correct. We have tried to address this problem by running all the tests in the image, to collect as much type information as possible.

Our choice to treat the assignment types of a variable to be truthful as they are, without considering the subtypes, may have influenced the results in class-based approach. If subtypes were to be considered as potential types, the results could be influenced in two ways: the number of correctly inferred variables might decrease due to the increased number of possible types for a variable, or they might increase if the run-time type of a variable is actually a subtype of the assigned type.

We have used only intra-procedural analysis in our algorithm. Application of inter-procedural analysis would certainly improve the results.

Another problem is that the run-time type information provides only a subset of the concrete types. For instance, if a method is present in Collection, and the current code uses it only in two subclasses, namely, OrderedCollection and Set, the type feedback will provide an array with OrderedCollection and Set as types and will provide neither the abstract type (Collection) nor the other concrete types coming from other subclasses, such as Array or Dictionary.

*External validity.* We have chosen projects *Roassal2*, *Glamour*, *Morphic* and *Moose* to evaluate ICTI since we were able to run these projects in a way that closely resembles their real usage. It is an open question whether we have collected all possible run-time types for variables.

We have chosen Pharo Smalltalk, since Smalltalk represents a "pure object-oriented, dynamically-typed language" while Pharo allows an easy implementation of the analysis tools. We cannot claim that the approach would retain the same efficiency in other dynamically-typed languages. When it comes to the dy-

namic type gatherer, theoretically, it may be constructed for any programming language that has a method-based runtime compiler, such as V8 for JavaScript [38]. However, we are not aware of the level of difficulty it would produce. There is a question of how to implement a dynamic type data gatherer for VMs using a meta-tracing runtime compiler, such as LuaJIT [35] and Pypy [36], as their execution model is different than the method-based one. As for the static code analysis, it may not be trivial to do in other object-oriented languages, due to the need for external tools to analyse the source code and obtain the AST of the code, *e.g.,* Moose [59] or Rascal [60].

## 8. Conclusion

Having type information at compile time can be useful when performing program maintenance tasks. In order to provide accurate information, type inference algorithms are often very complex, and still they suffer from the problem of false positives. On the other hand there exist lightweight algorithms that tend to work fast, and do not require whole program analysis, but which tend to be less precise.

In this paper we have presented a simple heuristic (ICTI) that aims to produce precise type information by using easily accessible information from inline caches. It collects the information about the frequency of class usage as receiver type from inline caches, and sorts statically inferred types based on this frequency. ICTI succeeds into promoting the correct type to the top of the list, whereas the basic algorithm produces unordered results. ICTI needs no instrumentation. It was evaluated using a prototype implemented in Pharo Smalltalk. We have focused our attention not only on inferring the root type of the variable, but also the correct subclass.

ICTI was able to increase the number of variables for which it correctly inferred types by more than 100% when compared with the basic approach.

*Acknowledgements*

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

We thank Eliot Miranda for helping us to implement the primitives we added in the Pharo VM and reviewing all our related commits.

## References


[1] S. Kleinschmager, S. Hanenberg, R. Robbes, E. Tanter, A. Stefik, Do static type systems improve the maintainability of software systems? An empirical study, in: 2012 IEEE 20th International Conference on Program Comprehension (ICPC), 2012, pp. 153 –162. doi:10.1109/ICPC.2012.6240483.

[2] S. Spiza, S. Hanenberg, Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study, in: Proceedings of the 13th International Conference on Modularity, MODULARITY '14, ACM, New York, NY, USA, 2014, pp. 99–108. doi:10.1145/2577080.2577098.
URL http://doi.acm.org/10.1145/2577080.2577098

[3] R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17 (1978) 348–375.

[4] O. Agesen, The Cartesian product algorithm, in: W. Olthoff (Ed.), Proceedings ECOOP '95, Vol. 952 of LNCS, Springer-Verlag, Aarhus, Denmark, 1995, pp. 2–26.

[5] P. Rapicault, M. Blay-Fornarino, S. Ducasse, A.-M. Dery, Dynamic type inference to support object-oriented reengineering in Smalltalk, in: Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS), 1998, pp. 76–77.
URL http://scg.unibe.ch/archive/famoos/Rapi98a/type.pdf

[6] M. Salib, Faster than C: Static type inference with Starkiller, in: in PyCon Proceedings, Washington DC, SpringerVerlag, 2004, pp. 2–26.

[7] S. A. Spoon, O. Shivers, Demand-driven type inference with subgoal pruning: Trading precision for scalability, in: Proceedings of ECOOP'04, 2004, pp. 51–74.

[8] F. Pluquet, A. Marot, R. Wuyts, Fast type reconstruction for dynamically typed programming languages, in: DLS '09: Proceedings of the 5th Symposium on Dynamic languages, ACM, New York, NY, USA, 2009, pp. 69–78. doi:10.1145/1640134.1640145.

[9] J. Davies, D. M. Germán, M. W. Godfrey, A. Hindle, Software bertillonage: Finding the provenance of an entity, in: MSR'11: Proceedings of the 8th International Working Conference on Mining Software Repositories, 2011, pp. 183–192. doi:doi.acm.org/10.1145/1985441.1985468.

[10] D. An, A. Chaudhuri, J. Foster, M. Hicks, Dynamic inference of static types for Ruby, in: Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11), ACM, 2011, pp. 459–472.

[11] E. Allende, O. Callaú, J. Fabry, É. Tanter, M. Denker, Gradual typing for Smalltalk, Science of Computer Programmingdoi:10.1016/j.scico.2013.06.006.
URL http://hal.inria.fr/hal-00862815

[12] M. P. Odgaard, JavaScript type inference using dynamic analysis, Master's thesis, Aarhus University (2014).
URL http://cs.au.dk/fileadmin/site_files/cs/Masters_and_diplomas/MortenPassowOdgaard.pdf

[13] A. Aiken, B. Murphy, Static type inference in a dynamically typed language, in: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '91, ACM, New York, NY, USA, 1991, pp. 279–290. doi:10.1145/99583.99621.
URL http://doi.acm.org/10.1145/99583.99621

[14] U. Hölzle, D. Ungar, Optimizing dynamically-dispatched calls with run-time type feedback, in: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, ACM, New York, NY, USA, 1994, pp. 326–336. doi:10.1145/178243.178478.

[15] N. Milojković, O. Nierstrasz, Exploring cheap type inference heuristics in dynamically typed languages, in: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, ACM, New York, NY, USA, 2016, pp. 43–56. doi:10.1145/2986012.2986017.
URL http://scg.unibe.ch/archive/papers/Milo16b.pdf

[16] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, D. Vardoulakis, In defense of soundiness: A manifesto, Commun. ACM 58 (2) (2015) 44–46.

doi:10.1145/2644805.
URL http://doi.acm.org/10.1145/2644805

[17] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Proceedings POPL '84, Salt Lake City, Utah, 1984. doi:10.1145/800017.800542.
URL http://webpages.charter.net/allanms/popl84.pdf

[18] U. Hölzle, C. Chambers, D. Ungar, Ecoop'91 european conference on object-oriented programming: Geneva, switzerland, july 15–19, 1991 proceedings, in: Proceedings of the European Conference on Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 21–38. doi:10.1007/BFb0057013.
URL http://dx.doi.org/10.1007/BFb0057013

[19] B. Foote, R. E. Johnson, Reflective facilities in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 327–336.

[20] N. Milojković, C. Béra, M. Ghafari, O. Nierstrasz, Inferring types by mining class usage frequency from inline caches, in: Proceedings of International Workshop on Smalltalk Technologies (IWST 2016), 2016, pp. 6:1–6:11. doi:10.1145/2991041.2991047.
URL http://scg.unibe.ch/archive/papers/Milo16a.pdf

[21] N. Milojković, A. Caracciolo, M. Lungu, O. Nierstrasz, D. Röthlisberger, R. Robbes, Polymorphism in the spotlight: Studying its prevalence in Java and Smalltalk, in: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, IEEE Press, 2015, pp. 186–195, published. doi:10.1109/ICPC.2015.29.
URL http://scg.unibe.ch/archive/papers/Milo15a.pdf

[22] O. Agesen, J. Palsberg, M. I. Schwartzbach, Type inference of SELF: Analysis of objects with dynamic and multiple inheritance, in: O. Nierstrasz (Ed.), Proceedings ECOOP '93, Vol. 707 of LNCS, Springer-Verlag, Kaiserslautern, Germany, 1993, pp. 247–267.
URL http://www.cs.purdue.edu/homes/palsberg/publications.html

[23] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983.
URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[24] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 227–242. doi:10.1145/38765.38828.

[25] J. Palsberg, M. I. Schwartzbach, Object-oriented type inference, in: Proceedings OOPSLA '91, ACM SIGPLAN Notices, Vol. 26, 1991, pp. 146–161.
URL http://www.cs.purdue.edu/homes/palsberg/publications.html

[26] N. Milojković, M. Ghafari, O. Nierstrasz, Exploiting type hints in method argument names to improve lightweight type inference, in: 25th IEEE International Conference on Program Comprehension, 2017. doi:10.1109/ICPC.2017.33.
URL http://scg.unibe.ch/archive/papers/Milo17a.pdf

[27] S. A. Spoon, O. Shivers, Dynamic data polyvariance using source-tagged classes, in: R. Wuyts (Ed.), Proceedings of the Dynamic Languages Symposium'05, ACM Digital Library, 2005, pp. 35–48.

[28] B. Spasojević, M. Lungu, O. Nierstrasz, Mining the ecosystem to improve type inference for dynamically typed languages, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! '14, ACM, New York, NY, USA, 2014, pp. 133–142. doi:10.1145/2661136.2661141.
URL http://scg.unibe.ch/archive/papers/Spas14c.pdf

[29] R. Cartwright, M. Fagan, Soft typing, in: PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 1991, pp. 278–292. doi:10.1145/113445.113469.

[30] B. Hackett, S.-y. Guo, Fast and precise hybrid type inference for JavaScript, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, New York, NY, USA, 2012, pp. 239–250. doi:10.1145/2254064.2254094.
URL http://doi.acm.org/10.1145/2254064.2254094

[31] S. Tobin-Hochstadt, V. St-Amour, The typed Racket guide, http://docs.racket-lang.org/ts-guide/.

[32] M. Furr, Combining static and dynamic typing in Ruby, Ph.D. thesis, University of Maryland (2009).
URL https://www.cs.umd.edu/~jfoster/papers/thesis-furr.pdf

[33] J. O. Graver, R. E. Johnson, A type system for Smalltalk, in: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90, ACM, New York, NY, USA, 1990, pp. 136–150. doi:10.1145/96709.96722.
URL http://doi.acm.org/10.1145/96709.96722

[34] R. Smit, Pegon, https://sourceforge.net/projects/pegon/.

[35] M. Pall, The LuaJIT Project, http://luajit.org/ (2005).

[36] C. F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, in: ICOOOLPS '09: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ACM, New York, NY, USA, 2009, pp. 18–25. doi:10.1145/1565824.1565827.

[37] M. Paleczny, C. Vick, C. Click, The Java hotspot$^{TM}$ server compiler, in: Proceedings of the 2001 Symposium on Java$^{TM}$ Virtual Machine Research and Technology Symposium - Volume 1, JVM'01, USENIX Association, Berkeley, CA, USA, 2001, pp. 1–1.
URL http://dl.acm.org/citation.cfm?id=1267847.1267848

[38] Google, Google's high performance, open source, javascript engine., https://developers.google.com/v8/ (2008).

[39] L. P. Deutsch, A. M. Schiffman, Efficient Implementation of the Smalltalk-80 system, in: Principles of Programming Languages, POPL '84, 1984. doi:10.1145/800017.800542.
URL http://webpages.charter.net/allanms/popl84.pdf

[40] U. Hölzle, C. Chambers, D. Ungar, Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, in: European Conference on Object-Oriented Programming, ECOOP '91, London, UK, UK, 1991.

[41] C. Bera, S. Ducasse, A. Bergel, D. Cassou, J. Laval, Handling exceptions, in: Deep Into Pharo, Square Bracket Associates, 2013, p. 38.

[42] A. Chiş, T. Gîrba, O. Nierstrasz, The Moldable Debugger: A framework for developing domain-specific debuggers, in: B. Combemale, D. J. Pearce, O. Barais, J. J. Vinju (Eds.), Software Language Engineering, Vol. 8706 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 102–121. doi:10.1007/978-3-319-11245-9_6.
URL http://scg.unibe.ch/archive/papers/Chis14b-MoldableDebugger.pdf

[43] K. Beck, Smalltalk Best Practice Patterns, Prentice-Hall, 1997.
URL http://stephane.ducasse.free.fr/FreeBooks/BestSmalltalkPractices/Draft-Smalltalk%20Best%20Practice%20Patterns%20Kent%20Beck.pdf

[44] M. Zandstra, PHP Objects, Patterns, and Practice, 4th Edition, Apress, Berkely, CA, USA, 2013.

[45] M. Bolin, Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript, O'Reilly Media, 2010.
URL https://books.google.ch/books?id=p7uyWPcVGZsC

[46] Z. Xu, X. Zhang, L. Chen, K. Pei, B. Xu, Python probabilistic type inference with natural language support, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 607–618. doi:10.1145/2950290.2950343.
URL http://doi.acm.org/10.1145/2950290.2950343

[47] C. Souza, E. Figueiredo, How do programmers use optional typing? An empirical study, in: Proceedings of the 13th International Conference on Modularity, MODULARITY '14, ACM, New York, NY, USA, 2014, pp. 109–120. doi:10.1145/2577080.2582208.
URL http://doi.acm.org/10.1145/2577080.2582208

[48] M. Faldborg, T. L. Nielsen, B. Thomsen, Type systems and programmers: A look at optional typing in Dart, Master's thesis, Aalborg University (2015).

[49] B. Spasojević, M. Lungu, O. Nierstrasz, A case study on type hints in method argument names in Pharo Smalltalk projects, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 283–292. doi:10.1109/SANER.2016.41.
URL http://scg.unibe.ch/archive/papers/Spas16a.pdf

[50] S. S. SHAPIRO, M. B. WILK, An analysis of variance test for normality (complete samples), Biometrika 52 (3-4) (1965) 591. arXiv:/oup/backfile/content_public/journal/biomet/52/3-4/10.1093/biomet/52.3-4.591/2/52-3-4-591.pdf, doi:10.1093/biomet/52.3-4.591.
URL +http://dx.doi.org/10.1093/biomet/52.3-4.591

[51] P. Bunge, Scripting browsers with Glamour, Master's thesis, University of Bern (Apr. 2009).
URL http://scg.unibe.ch/archive/masters/Bung09a.pdf

[52] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, J. Laval, Agile visualization with Roassal, in: Deep Into Pharo, Square Bracket Associates, 2013, pp. 209–239.

[53] H. Fernandes, S. Stinckwich, Morphic, les interfaces utilisateurs selon Squeak (Jan. 2007).

[54] T. Gîrba, The Moose book (2010).
URL http://www.themoosebook.org/book

[55] S. Ducasse, T. Gîrba, M. Lanza, S. Demeyer, Moose: a collaborative and extensible reengineering environment, in: Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, Franco Angeli, Milano, 2005, pp. 55–71.
URL http://scg.unibe.ch/archive/papers/Duca05aMooseBookChapter.pdf

[56] S. Ducasse, M. Lanza, S. Tichelaar, Moose: an extensible language-independent environment for reengineering object-oriented systems, in: Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools), 2000.
URL http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf

[57] O. Callaú, R. Robbes, É. Tanter, D. Röthlisberger, A. Bergel, On the use of type predicates in object-oriented software: The case of Smalltalk, in: Proceedings of the 10th ACM Dynamic Languages Symposium (DLS 2014), ACM Press, Portland, OR, USA, 2014, pp. 135–146. doi:10.1145/2661088.2661091.
URL http://pleiad.dcc.uchile.cl/papers/2014/callauAl-dls2014.pdf

[58] D. Thomas, C. Fowler, A. Hunt, Programming Ruby 1.9: The Pragmatic Programmers' Guide, 3rd Edition, Pragmatic Bookshelf, 2009.

[59] S. Ducasse, T. Gîrba, O. Nierstrasz, Moose: an agile reengineering environment, in: Proceedings of ESEC/FSE 2005, 2005, pp. 99–102, tool demo. doi:10.1145/1081706.1081723.
URL http://scg.unibe.ch/archive/papers/Duca05fMooseDemo.pdf

[60] M. Hills, P. Klint, J. J. Vinju, Scripting a refactoring with rascal and eclipse, in: Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12, ACM, New York, NY, USA, 2012, pp. 40–49. doi:10.1145/2328876.2328882.
URL http://doi.acm.org/10.1145/2328876.2328882